Assignment 1: web_crawler, Web Crawler

This is the final draft of the assignment specification.
Additional clarifications and testing information may be added (as well as in the course forum). In any case, the changes will be indicated with different colours so you can spot them easily.
The core tasks of the assignment will not change

Assignment Files

You can obtain the assignment template files two ways:

- On a CSE server do:
  `$ 9024 fetch web_crawler`
  This will create a directory called 'web_crawler' with all the assignment start-up files.
- On a browser, copy/paste the following URL:
  `http://www.cse.unsw.edu.au/~cs9024/web_crawler.zip`

You can also make note of the following URL:
`http://www.cse.unsw.edu.au/~cs9024/mini-web`
Once you read the spec, hopefully it will be clear to you what you could use this URL for!

Background

As we have mentioned in lectures, the Internet can be though of as a graph (a very large graph).
Webpages represent vertices and hyperlinks represents directed edges.

With over 1.2 Billion unique websites (as of June 2021) and each website having multiple webpages and each webpage having multiple hyperlinks it can understandably be a very difficult job to remember the URL of every website you want to visit.

In order to make life easier, from the very early days of the internet, there have been search engines that can be used to find websites.

But the job of a search engine is very difficult:
First it must index (create a representation of) the entire (or as close to it as possible) internet.
Next it must rank the webpages it finds.

In this assignment we will be implementing algorithms to solve each of these problems.

First: To index the internet we will be creating a web crawling.
Second: To rank webpages we will implement the pagerank algorithm.
Third: To find the shortest path between two pages we will implement dijkstra's algorithm

The assignment

Subset 0 - Dependencies

Before we can start crawling we need to be able to store our crawled data.
As the internet is a Graph, this means we need a Graph ADT.
We will also need a Set ADT and one of a Queue ADT or a Stack ADT in order to perform web scraping (for a BFS or DFS).

Subset 0a - List (Queue, Stack, Set) ADT

Task

You have been provided with a file *list.h*.
list.h provides the interface to an ADT that will provide Queue, Stack, and Set functionality.

You should create the file *list.c* to implement this ADT.
Your list ADT should store *string* (*char **) data.

You can use whatever internal representation you like for your list ADT.

You can use code previously submitted for this course in your list ADT.

As a reminder:
Queue - First In, First Out
Stack - First In, Last Out

Stack - First In, Last Out

Set - Only stores unique values.

See *list.h* for more information about each function that you are required to implement.
You cannot modify *list.h*.

You should write programs that use the list ADT to test and debug your code.

Subset 0b - Graph ADT

Task

You have been provided with a file *graph.h*.
graph.h provides the interface to an ADT that will provide Graph functionality.

You should create the file *graph.c* to implement this ADT.
Your graph ADT should store *string* (*char **) data in its vertices.

You must use an adjacency list representation for your ADT.
But the exact implementation is up to you.

You can use code previously submitted for this course in your graph ADT.

See *graph.h* for more information about each function that you are required to implement.
Note that *graph.h* indicates that each edge has a weight.
You cannot modify *graph.h*.

You should write programs that use the graph ADT to test and debug your code.

Subset 1 - Web Crawler

Task

> WARNING:
>
> *crawler.c* requires external dependencies (*libcurl* and *libxml2*)
> The provided makefile will work on CSE servers (ssh and vlab) but might not work on your home computer.

We are now going to use the *list* and *graph* ADTs you have created to implement a web scraper.

If you have correctly implemented these ADTs, this part should be mostly free.

crawler.c is a complete implementation of a web crawler; you do not need to modify it for this particular Subset1 task. You should look at the program carefully and understand it well so that you can use it (i.e., modify it appropriately) for later tasks though.

Subset 2 - PageRank

Background

Now that we can crawl a network and build a graph, we need a way to determine what pages in our network (vertices) are important.

We haven't kept page content so the only metric we can use to determine importance of a page is to check how much other pages rely on its existence.
That is: how easy is it to follow a sequence of one or more links (edges) and end up on the page.

In 1998 Larry Page and Sergey Brin created the PageRank algorithm to evaluate this metric.

Google still uses the PageRank algorithm to score every page it indexes on the internet to help order its search results.

Task

In *graph.c* implement the two new functions `graph_pagerank()` and `graph_viewrank()`.

`graph_pagerank()` should calculate the PageRank of every page (vertex) in your graph and store them in the ADT.

`graph_viewrank()` should print the PageRank of every page (vertex) in your graph.
pages (vertices) should be printed from highest to lowest rank, if two pages have the same rank then they should be printed alphabetically.

> NOTE:
>
> You **can** add utility functions to *graph.c*
> You **can** (and most likely will need to) modify your struct definitions in *graph.c*
> You **cannot** modify the file *graph.h*
> You **cannot** modify the function prototypes for `graph_pagerank()` and `graph_viewrank()`

Algorithm

For $t = 0$:

$$PR(p_i; t) = \frac{1}{N}$$

for $t > 0$:

$$PR(p_i; t) = \frac{1 - d}{N} + d \times ((\sum_{p_j \in M(p_i)} \frac{PR(p_j; t-1)}{D(p_j)}) + (\sum_{p_j \in S} \frac{PR(p_j; t-1)}{N}))$$

Where:

N is the number of vertices
p_i and p_j are each some vertex
t is the "time" (iteration count)
$PR(p_i; t)$ is the PageRank of vertex p_i at some time t
d is the damping_factor
$M(p_i)$ is the set of vertices that have an outbound edge towards $M(p_i)$
$PR(p_j; t-1)$ is the PageRank of vertex p_j at some time $t-1$
$D(p_j)$ is the degree of vertex p_j, ie. the number of outbound edges of vertex p_j
S is the set of sinks, ie. the set of vertices with no outbound edges, ie. where $D(p_j)$ is 0

This is formula is equivalent to the following algorithm:

```
procedure graph_pagerank(G, damping_factor, epsilon)
    N = number of vertices in G
    for all V in vertices of G
        oldrank of V = 0
        pagerank of V = 1 / N
    end for
    while |pagerank of V - oldrank of V| of any V in vertices of G > epsilon
        for all V in vertices of G
            oldrank of V = pagerank of V
        end for
        sink_rank = 0
        for all V in vertices of G that have no outbound edges
            sink_rank = sink_rank + (damping_factor * (oldrank of V / N))
        end for
        for all V in vertices of G
            pagerank of V = sink_rank + ((1 - damping_factor) / N)
            for all I in vertices of G that have an edge from I to V
                pagerank of V = pagerank of V + ((damping_factor * oldrank of I) / number of outbound edges
from I)
            end for
        end for
    end while
end procedure
```

In order to test your pagerank functions, you should modify *crawler.c* to add *#include "pagerank.h"*, and change the last part of the main function to something like

```
    graph_show(network, stdout);
    graph_pagerank(network, damping, delta);
    graph_viewrank(network, stdout);
    graph_destroy(network);
```

where you choose appropriate values for *damping* and *delta* (aka epsilon).

The changes you make to *crawler.c* are purely for you to test whether your pagerank functions are working. We use a different *crawler.c* for testing your pagerank functions.

Subset 3 - Degrees of Separation

Task

In *graph.c* implement the two new functions `graph_shortest_path()` and `graph_view_path()`.

`graph_shortest_path()` should calculate the shortest path between a source vertex and all other vertices.
`graph_shortest_path()` should use dijkstra's algorithm to do so.
Unlike a regular implementation of dijkstra's algorithm, your code should minimise the number of edges in the path (not minimise the total weight of the path).

`graph_view_path()` should print the path from the previously given source vertex to a given destination vertex.

> NOTE:
>
> You **can** add more utility functions to *graph.c*
> You **can** (and most likely will need to) extend your struct definitions in *graph.c*
> You **cannot** modify the file *graph.h*
> You **cannot** modify the function prototypes for `graph_shortest_path()` and `graph_view_path()`

In order to test your dijkstra functions, you should modify *crawler.c* to add *#include "dijkstra.h"*, and change the last part of the main function to something like:

```
    graph_show(network, stdout);
    graph_shortest_path(network, argv[1]);
    char destination[BUFSIZ];
    printf("destination: ");
    fgets(destination, BUFSIZ, stdin);
    destination[strlen(destination)-1] = '\0'; // trim '\n'
    graph_view_path(network, destination);
    graph_destroy(network);
```

The changes you make to *crawler.c* are purely for you to test whether your dijkstra functions are working. We use a different *crawler.c* for testing your dijkstra functions.

Assessment

Testing

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 9024 autotest web_crawler
```

`9024 autotest` will not test everything.
Always do your own testing.

Automarking will be run by the lecturer after the submission deadline, using a superset of tests to those `autotest` runs for you.

Submission

When you are finished working on the assignment, you must submit your work by running `give`:

```
$ give cs9024 ass1_web_crawler list.c graph.c
```

You must run `give` before **6 August 21:59:59** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times.
Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

You *cannot* obtain marks by e-mailing your code to tutors or lecturers.

You can check your latest submission on CSE servers with:

```
$ COMP(9024|9024) classrun -check ass1_web_crawler
```

You can check the files you have submitted here.

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can view your results here; The resulting mark will also be available via give's web interface.

Due Date

Due Date

This assignment is due **6 August 21:59:59**.

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 2%. For example, if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 15 hours late, it would be awarded 70%, the maximum mark it can achieve at that time.

Assessment Scheme

This assignment will contribute 12 marks to your final COMP(9024|9024) mark

20% of the marks for this assignment will come from hand-marking. These marks will be awarded on the basis of clarity, commenting, elegance and style: in other words, you will be assessed on how easy it is for a human to read and understand your program.

80% of the marks for this assignment will come from the performance of your code on a large series of tests.

An indicative assessment scheme follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

HD (85+)	All subsets working; code is beautiful;
DN (75+)	Subset 2 working; good clear code;
CR (65+)	Subset 1 working; good clear code;
PS (55+)	Subset 0 passing some tests; code is reasonably readable;
PS (50+)	Good progress on assignment, but not passing autotests
0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP(9024|9024)	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Attribution of Work

This is an individual assignment.

The work you submit must be entirely your own work, apart from any exceptions explicitly included in the assignment specification above. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

You are only permitted to request help with the assignment in the course forum, help sessions, or from the teaching staff (the lecturer(s) and tutors) of COMP(9024|9024).

Do not provide or show your assignment work to any other person (including by posting it on the forum), apart from the teaching staff of COMP(9024|9024). If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if that work was submitted without your knowledge or consent; this may apply even if your work is submitted by a third party unknown to you. You will not be penalized if your work is taken without your consent or knowledge.

Do not place your assignment work in online repositories such as github or any where else that is publically accessible. You may use a private repository.

Submissions that violate these conditions will be penalised. Penalties may include negative marks, automatic failure of the course, and possibly other academic discipline. We are also required to report acts of plagiarism or other student misconduct: if students involved hold scholarships, this may result in a loss of the scholarship. This may also result in the loss of a student visa.

Assignment submissions will be examined, both automatically and manually, for such submissions.

Change Log